ChatGPT

Dirty Flags and Incremental Computation in Dependency Updates

Understanding Dirty Flags

A **dirty flag** (or dirty bit) is a boolean marker indicating that a piece of derived data is "out-of-date" relative to its source data [1]. In practice, it means the value has changed (or become "dirty") and needs to be recomputed or synchronized. For example, in a game engine's scene graph, an object's world transform might have a dirty flag set whenever its local transform changes; when the world transform is needed, the system checks the flag and recalculates the transform only if it's dirty [2]. This pattern avoids unnecessary work by **only recalculating derived values when their inputs have actually changed** [3]. Once the recalculation is done, the flag is cleared until the next change.

In summary, a dirty flag serves as a simple form of cache invalidation: it tracks whether a cached value (derived data) is invalid due to changes in its inputs (primary data). If the flag is set, we know the cache is stale and must recompute; if not, we can reuse the previous result [1]. This concept is key to implementing **incremental computation**, where we update only the parts of a system that have actually changed, rather than recomputing everything from scratch on each update.

Incremental Computation vs. Full Recalculation

Incremental computation means updating results incrementally in response to changes, instead of recalculating the entire system every time. In a dependency graph (like nodes representing computations with input/output relationships), an incremental approach will recompute only those nodes affected by a change. Nodes whose inputs have not changed can skip work, using their cached values from the previous computation.

Contrast this with a naive approach where every node is recomputed every time (full recalculation). The benefit of incremental update is performance: by **"only recalculating cells dependent on changed cells,"** a lot of unnecessary computation is avoided [4]. For instance, a spreadsheet like Lotus 1-2-3 in the 80s would recalc all formulas in natural dependency order, but more advanced spreadsheets (e.g. SuperCalc5) realized they could skip formulas whose precedents didn't change [5]. This saved time especially in large documents. Similarly, game engines or simulation systems use dirty flags to avoid re-updating static objects that haven't moved or changed [6] [7].

However, to do incremental computation correctly, we must update nodes in the **right order** when multiple inputs change. If two or more inputs change simultaneously, their dependent nodes must be evaluated in a sequence that respects dependencies (usually a topological order). Otherwise, you risk using stale values. For example, if input A and a downstream value Z both change at once, you should recompute A's dependents before Z's, or else Z might be computed using the old value of A [8]. Getting the order wrong

can lead to extra computations (recomputing Z twice) or incorrect results. This is why dependency graphs are often processed in topologically sorted order.

Using Dirty Flags for Incremental Updates

To incorporate dirty flags and incremental recomputation into the evaluation engine, we can follow a strategy like this:

1. **Maintain a Changed/Dirty Status for Nodes:** Keep a boolean flag for each node indicating whether its output value has changed in the current update cycle. This could be stored in a separate parallel array of flags or as a field in each node object. Initially, no nodes are dirty.

2. **Mark Initial Changes:** At the start of an update (e.g. a new "tick" or input event), identify which input nodes have externally changed values. Mark those nodes as dirty, since their outputs are now out of sync and need propagation. These are the starting points for recomputation.

3. **Topologically Sorted Evaluation:** Iterate through the nodes in **dependency order** (the array is already sorted such that each node comes after all of its input dependencies). For each node:

4. **Check Dependencies:** Determine if this node needs to update by checking its inputs' dirty flags. If none of its input values changed (i.e., all inputs are clean in this cycle), then this node's output remains valid and **no recomputation is needed** – you can skip evaluating it this round.
5. **Recompute If Needed:** If at least one input is marked dirty (meaning an input's value changed), then evaluate this node's function to produce a new output value.
6. **Detect Output Change:** After evaluation, compare the node's new output with its previous output (from the last cycle). If the value is different (you may use an exact comparison or a tolerance threshold for floating-point values as appropriate), then mark this node itself as **dirty (changed)**. This signals that dependent nodes downstream should be recalculated. If the output remains effectively unchanged, set this node's dirty flag to false (or "clean"), meaning no significant change occurred.

7. This per-node process can often be implemented by having the node's `eval()` method return a boolean indicating whether its output changed. Alternatively, the node can set an internal "outputChanged" bit that the engine reads. The key is that **each node "decides" if its value changed enough to propagate** (for example, setting its dirty flag or returning true on change).

8. **Propagate Changes in One Pass:** Because the nodes are processed in a sorted (topologically ordered) sequence, by the time you reach a given node, all of its prerequisite inputs have already been updated and flagged if changed. Thus, you can safely use their dirty flags to decide whether to recompute the current node. This allows the entire dependency graph to update **in a single forward pass** following the topological order, without needing a separate pre-marking phase or multiple sweeps. Essentially, the change propagation and computation happen together: as you go through the list, you compute only nodes that are affected by earlier changes, and skip the rest.

9. *Contrast with two-phase approach:* An alternative approach is to first do a "dirty marking sweep" – traverse the dependency graph and mark all downstream nodes of any changed input as dirty, then in a second phase, perform recomputation of those dirty nodes in the correct order [9]. While that

method works, it can mark many nodes as dirty even if some ultimately don't change in value. It also involves extra overhead in setting/clearing flags. By merging the marking and recompute into one pass, we simplify the process. As long as the evaluation order is correct (ensuring inputs are processed before outputs), **separate marking and then computing isn't strictly necessary for correctness** – we can do it in one integrated traversal, which is exactly what you propose.

10. **Result:** After this pass, all nodes that depend (directly or transitively) on the changed inputs will have been recalculated exactly once, and all outputs will be consistent with the new inputs. Nodes unaffected by the change were never evaluated, saving computation. Each input change thus incurs recomputation only along the paths that originate from that input, rather than a full graph recompute. This approach guarantees that **"no cell is ever recalculated unless one of its inputs actually changed,"** which is a big efficiency win [10].

Illustration: In a spreadsheet dependency graph, when a cell (node) changes, all cells downstream of it can be marked as "dirty" (orange tags) to indicate they need recalculation. An incremental algorithm will then recompute those dirty cells in the correct order (ensuring prerequisites are computed before dependents) and skip any cells that remain clean. This way, only the affected portions of the graph are updated [9] [11].

Implementation Details and Tips

- **Storing Previous Values:** To know if a node's output has "changed sufficiently," the node should store its last computed value. After computing a new value, compare it to the stored value. If different (beyond a tolerance if needed), update the stored value and mark the node as changed. If it's the same, you can leave the stored value as is and consider the node clean (no change). Storing the last value makes change detection straightforward.

- **Threshold for Change:** Depending on the domain, you might not want to propagate very tiny changes (for example, due to floating-point precision noise). A node's `eval()` could incorporate a threshold to decide if the difference between new and old output is significant enough to count as a change. If not, the node can treat it as "no change" (dirty flag stays false), and thereby avoid unnecessarily triggering downstream updates on minor fluctuations.

- **Dirty Flag Array vs. Node Property:** Using a separate boolean array (parallel to the node list) to track dirty status is convenient for an indexed implementation. For example, `dirty[i]` could represent whether node `i` changed in the current cycle. This array gets updated as you compute each node. Alternatively, each node object can carry its own `dirty` field or a method to query if it changed. Both approaches are valid; what matters is that by the time you process the next node, you can efficiently check if any of its input nodes were marked dirty.

- **Marking Inputs:** The "outside" engine or evaluator needs to interface with whatever is providing input changes. For instance, if certain input nodes are fed by external data or user input, when those change, the engine should set those nodes' dirty flags true (and possibly update their values) before running the evaluation pass. This initialization step ensures the change propagation starts correctly.

- **Single-Pass Propagation:** As noted, you can perform the dirty checking and recomputation in one pass through the sorted nodes. This works because the sorting guarantees that when a node is

reached, all of its dependencies have already been processed. If any dependency changed and was marked dirty, this node will see that flag and know it needs updating. There's no need for an upfront sweep to mark transitive dependencies; the information percolates naturally as you iterate. This is efficient and simpler to implement. Each "input change event" effectively triggers one evaluation pass over the nodes. If multiple inputs change at once (in the same event or time step), mark all of them dirty before the pass – the algorithm will handle all their downstream effects together. If changes arrive one at a time (multiple sequential events), you would run a pass for each event, or potentially batch them if appropriate. But you **do not** repeatedly loop until convergence or anything, because the topological order ensures one pass resolves all direct consequences of those input changes (assuming no cycles in the graph).

- **Correctness Considerations:** Make sure that your dependency graph truly has a topologically sorted order (no cycles). In a cyclic graph, incremental propagation is more complex and may require iterative approaches or cycle detection. But if the graph is acyclic and sorted, one pass is sufficient to propagate all changes. Also, ensure that if a node's output does not actually change (value remains equal), you do **not** mark it dirty – this prevents unnecessary downstream recalculations. This was a known issue with a simple dirty-marking strategy: you might mark a whole chain dirty even if a recomputed node ends up with the same value, causing needless work [9]. By checking for real changes, we avoid that pitfall.

Benefits of This Approach

By incorporating dirty flags and an incremental update strategy, the system gains significant efficiency improvements: - **Skip Unchanged Calculations:** Nodes whose inputs haven't changed are skipped entirely, saving time. As noted, this means we "only recalc cells dependent on changed cells" rather than everything [4]. - **Minimal Recomputations:** Each affected node is recalculated at most once per input change event (assuming no cycles), thanks to proper ordering. We eliminate redundant recalculations. In the ideal case, a changed value and all of its downstream dependents are each computed just one time [12]. - **Dynamic Adaptivity:** The system naturally adapts to changes of varying scope. A small change (affecting only a few nodes) triggers only a small amount of recomputation, whereas a big change will propagate more – but you pay for what you use. This is much more scalable than always doing a full evaluation. - **Real-World Examples:** Modern spreadsheets like Excel use a combination of dirty-flag marking and topological ordering to update formulas efficiently [13]. Game engines use dirty flags for things like rendering transformations so that static objects aren't recalculated every frame [7]. These techniques show that carefully tracking "what changed" can greatly improve performance in large systems.

Conclusion

Dirty flags provide a mechanism for nodes to determine if their outputs are out-of-date, and **incremental computation** ensures that only the necessary recomputations are performed after a change. By letting each node signal whether it has *actually* changed (via a boolean flag or similar) and processing the graph in dependency order, we achieve an efficient update loop: the evaluator marks changed inputs dirty, then walks through the sorted nodes, recomputing only what's needed and propagating change flags downstream. This integrated one-pass approach handles change propagation without a separate marking phase, unless a particular scenario demands it for correctness (which in a DAG it typically does not). The result is a system that updates correctly and much faster than naive full recomputation, because

unchanged parts of the graph simply remain untouched – a clear win for performance and scalability in complex evaluations.

Sources: Dirty flag pattern definition [1] [2]; Incremental recomputation strategies in spreadsheets [4] [10]; Dirty marking vs. topological update trade-offs [9] [11].

---

[1] [2] [6] [7] [12] Dirty Flag · Optimization Patterns · Game Programming Patterns
https://gameprogrammingpatterns.com/dirty-flag.html

[3] [4] [5] [8] [9] [10] [11] [13] How to Recalculate a Spreadsheet – Lord.io
https://lord.io/spreadsheets/